Creación de una demo 3D mediante gráficos procedurales

Memoria del Proyecto Final de Carrera de Ingeniería Informática
Realizado por David Rodríguez Real
y dirigido por Enric Martí Gòdia
Bellaterra, 19 de junio de 2012

El abajo firmante, Enric Martí Gòdia

Profesor de la "Escola d'Enginyeria" de la UAB,

CERTIFICA:

Que el trabajo al que corresponde esta memoria ha sido realizado bajo su dirección por **David Rodríguez Real**

Y para que conste firma la presente.

Firmado: Enric Martí Gòdia

Bellaterra, 19 de junio de 2012

Agradecimientos

Quiero agradecer a toda mi familia por su continuo apoyo durante este duro proceso. Así como a mis profesores y tutores del proyecto, por haber tenido paciencia conmigo.

Y por supuesto, agradecer a todos mis amigos y compañeros de la universidad por haber estado ahí durante este largo periplo por la carrera: Cristóbal, Felipe, Jordi, Rubén, Jon, Adrián, Arnau, Víctor, Iván, Julián, y tantos otros que me dejaré.

Índice

Capítulo 1

Introducción

Entendemos como "procedural" todo aquel elemento que es generado por un procedimiento o código informático. Ésta definición abarca diferentes campos, especialmente en el área de los videojuegos [1] y el cine. En el campo de los gráficos por computador hablamos de gráficos procedurales para referirnos a todo aquel elemento visual que es generado directamente por un algoritmo en vez de ser creado manualmente con herramientas externas[1].

Si bien a primera vista no parece haber diferencia en el resultado final, los gráficos procedurales disponen de una gran ventaja respecto a los que están hechos a mano: se pueden generar muchos con muy poco. En otras palabras, mientras que los gráficos tradicionales requieren de un intenso trabajo previo por parte del artista gráfico (más aún actualmente, ya que se ha llegado a un alto grado de realismo y sofisticación) resultando en un modelo singular, los procedurales, una vez debidamente implementados, pueden generar amplios resultados con solo variar unos pocos parámetros. Es precisamente ese su talón de Aquiles: conseguir implementar un efecto gráfico procedural en base a los estándares de hoy en día no es nada sencillo. Para conseguir un modelo realmente detallado resulta más barato crearlo a mano que conseguir un algoritmo que lo genere automáticamente. Sin embargo, la generación procedural tiene otros usos más productivos.

Su auge en el mundo de los videojuegos empezó por la década de los 80. Los juegos de ordenador de aquella época sufrían de una limitada memoria, lo cual no permitía grandes alardes visuales. El surgimiento de diferentes técnicas procedurales permitió a los desarrolladores generar una gran cantidad de escenarios con los pocos medios que tenían, consiguiendo cifras que no se habían visto antes en la industria del videojuego. Casos como el de *The Sentinel* (figura 1.1.a), que permitía jugar hasta 10.000 niveles diferentes, o el más extremo de *Elite* (figura 1.1.b), donde se podían generar hasta 2^{48} galaxias de forma aleatoria con 256 sistemas solares en cada una, fueron recibidos con gran sorpresa.

[1] Software de diseño 3D como *3D Studio*, *Maya* o *Softimage*.

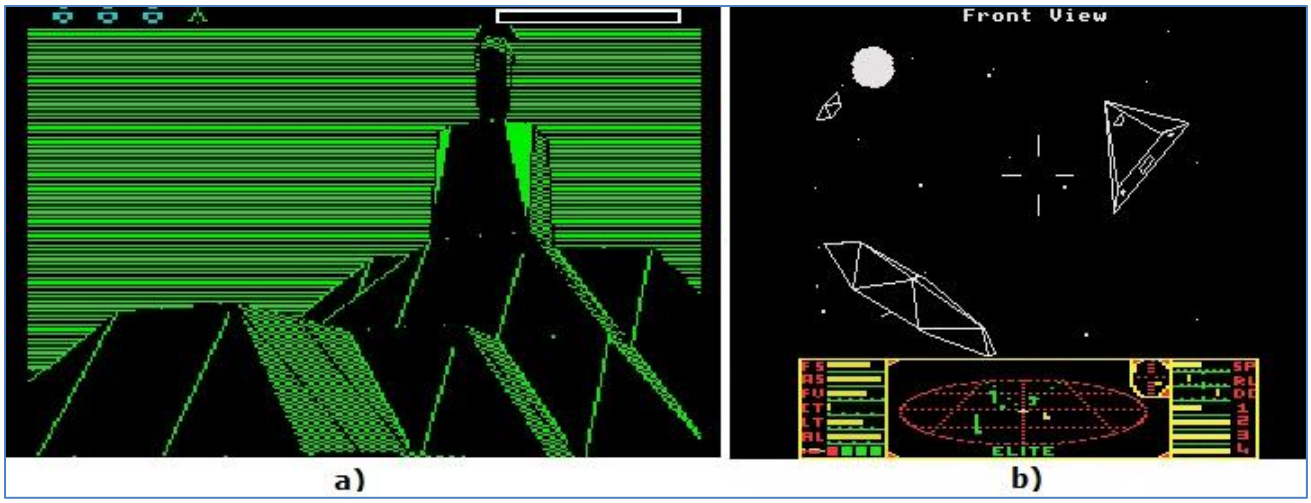

Fig. 1.1 *The Sentinel* (1986) y *Elite* (1984)

Paralelamente, por aquella época surgió una subcultura derivada de la cultura hacker: la *demoscene* [2]. Consistía en crear aplicaciones gráficas no-interactivas llamadas "demos" usando una gran variedad de efectos procedurales, llevando hasta el límite las capacidades de la máquina. Pronto se convirtió en una forma de competición que se sigue celebrando hasta ahora. Debido a la mayor potencia de los ordenadores actuales, los desarrolladores de la *demoscene* optan por plantearse nuevas metas: un estilo visual más atractivo y limitaciones en el tamaño del programa o la plataforma utilizada. Uno de los exponentes más conocidos de este movimiento es el mini-juego de disparos en primera persona *.kkrieger* (figura 1.2), el cual funciona en un ejecutable de tan solo 56 KBytes.

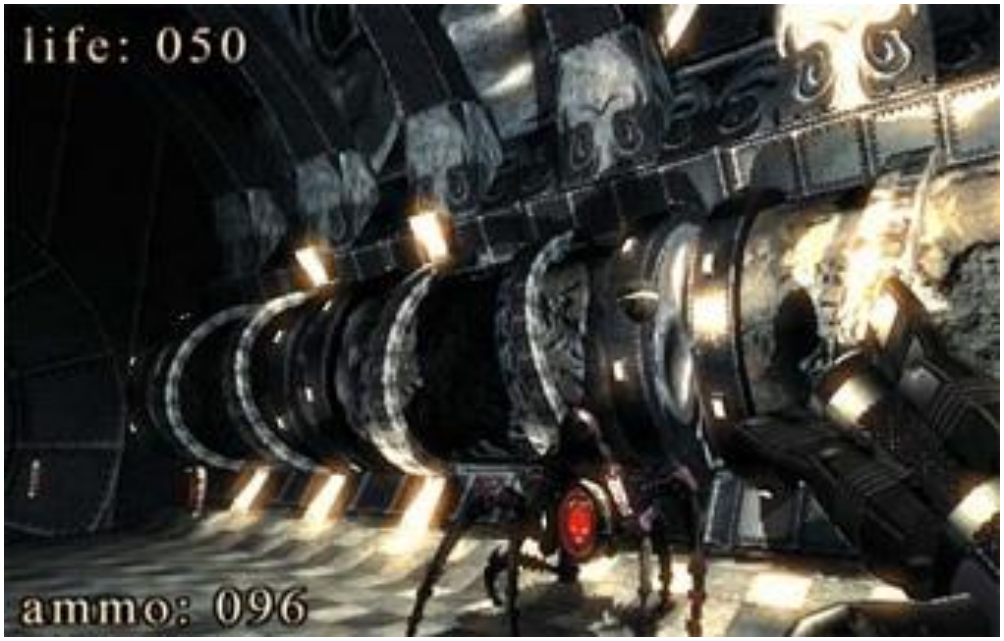

Fig. 1.2 Captura de *.kkrieger* (2004)

Hoy en día, los gráficos 3D han evolucionado y alcanzado un nivel de realismo muy elevado. Los videojuegos actuales tienen costes millonarios y una elevada cantidad de personal trabajando en su realización, muchos de ellos artistas gráficos. Debido a que actualmente la capacidad de memoria no es un problema, sale más a cuenta contratar a una gran cantidad de grafistas y modeladores para crear gráficos híper-detallados, más atractivos de cara al público. Es por ello que las técnicas procedurales gráficas cayeron parcialmente en desuso, dado que resultaba más barato el acercamiento tradicional. No obstante, el hecho de que los costes de desarrollo sean tan elevados está llevando a los desarrolladores de videojuegos a reducir costes como sea, ya sea a través del *outsourcing* o el uso de *middleware* para tareas concretas. Es ahora cuando los gráficos procedurales cobran relevancia de nuevo, ya que permiten ahorrar trabajo a los diseñadores dada su capacidad de "generar mucho con muy poco".

Este proyecto nace del deseo por experimentar con esta peculiar forma de generar gráficos por ordenador. El objetivo será, por tanto, crear una demo 3D utilizando exclusivamente técnicas procedurales, al estilo de la *demoscene*, pero dotándola de un mínimo de interactividad. De esta forma se podrá evaluar cómo de viable es esta forma de generar gráficos aplicada a videojuegos u otras aplicaciones multimedia. Dicha demo tendrá las siguientes características:

- Al ser creada exclusivamente con técnicas procedurales, se auto-impone que en la ejecución de la demo no se cargará información desde archivos externos. Todo lo que se genere será dentro del propio código del programa.

- La demo cargará un escenario exterior generado de forma aleatoria cada vez que el programa se ejecute, así como otros efectos adicionales.

- Como no sabemos de antemano el rendimiento que obtendremos al aplicar todos los efectos, reduciremos la interactividad a poder explorar el escenario en cualquier dirección, ya que este se generará en tiempo real.

En el **capítulo 2** se expondrán los requerimientos de nuestro proyecto, un estudio de viabilidad del mismo y una planificación preliminar de tareas a cumplir.

En el **capítulo 3** se detalla el proceso seguido en el desarrollo de la aplicación para implementar los diferentes elementos de la misma, así como los problemas surgidos por el camino y las decisiones tomadas para solventarlos.

En el **capítulo 4** veremos los resultados obtenidos tras finalizar la demo, los hitos conseguidos y las posibles mejoras a lo ya obtenido.

En el **capítulo 5** se discuten los resultados obtenidos, se hace una valoración global del trabajo en el proyecto y se expone una visión de futuro respecto al tema tratado.

En el **anexo A** se explican diferentes conceptos y algoritmos que se aplicarán a la hora de desarrollar nuestra demo, por lo que se les dedica una sección aparte.

En el **anexo B** se expone un breve manual de usuario para poder utilizar la aplicación correctamente.

Capítulo 2

Análisis

En este capítulo se definirán las **especificaciones del proyecto**, lo cual incluye los **requerimientos funcionales y no funcionales** de la aplicación y los **requerimientos técnicos** (*hardware* y *software* necesario). Se detallan las **herramientas utilizadas**. También se incluye un **estudio de viabilidad** y se expone una **planificación preliminar** del proyecto.

2.1 Especificaciones del proyecto

A continuación se detallarán los requerimientos funcionales (los que definen las funcionalidades de la aplicación) y los no funcionales (más técnicos y no enfocados de cara al usuario). Adicionalmente, se incluyen los posibles requerimientos técnicos para ejecutar la aplicación satisfactoriamente.

2.1.1 Requerimientos funcionales

Nuestra aplicación 3D tendrá una serie de objetivos a cumplir:

- La demo generará un terreno con forma aleatoria diferenciable cada vez que se ejecute el programa.
- Se incluirá una opción para cambiar la "semilla" de nuestro generador durante la ejecución y así modificar en tiempo real la disposición aleatoria del terreno.
- El usuario será capaz de moverse libremente por el escenario, de forma que el terreno procedural se vaya generando a su paso.
- En la demo se visualizarán efectos procedurales adicionales al terreno, como por ejemplo texturas procedurales.

2.1.2 Requerimientos no funcionales

De cara al desarrollo, se tendrán en cuenta los siguientes requerimientos:

- **Uso de las librerías de *DirectX*.** Para desarrollar una aplicación gráfica de este tipo se tienen dos opciones: utilizar *OpenGL* o *DirectX*. Se ha optado por este último ya que se tiene más experiencia previa con él, aunque la primera opción hubiera resultado igualmente viable. La versión de *DirectX* utilizada es la 10.
- **Desarrollado en C++ [3].** Es uno de los lenguajes más eficientes para aplicaciones en tiempo real y el más adecuado para trabajar con *DirectX*.
- **Sin cargas de archivos.** Como ya se ha comentado en la introducción, la idea es crear una demo completamente procedural al estilo de las de la *demoscene*. Por tanto, se impone la norma de no cargar ningún archivo desde disco a la hora de iniciar la demo sino generar todos los datos necesarios en tiempo de ejecución.
- **El realismo no es una prioridad.** La intención de la demo es implementar diferentes técnicas procedurales y evaluar su flexibilidad y rendimiento respecto al método tradicional. Por tanto, no nos centraremos en una alta fidelidad visual.

2.1.3 Requerimientos técnicos

Debido a que utilizaremos la versión 10 de DirectX, necesitaremos un ordenador con sistema operativo Windows Vista o Windows 7. Además, necesitamos que incorpore una tarjeta gráfica que soporte dicha versión de la API. Como no sabemos de antemano el rendimiento que obtendremos de la demo no se puede especificar un hardware mínimo. Sin embargo, con un PC de sobremesa de gama media debería bastar para nuestro propósito. El ordenador utilizado para desarrollar y depurar la demo contiene un *Intel I7 950* con una *GeForce GTX 460*.

2.2 Herramientas utilizadas

A continuación, se hará un breve estudio de las herramientas que hemos decidido utilizar para llevar a cabo este proyecto. Son las siguientes: *DirectX 10, Visual Studio 2008, Notepad++* y *FX Composer*.

2.2.1 DirectX 10

DirectX es una colección de librerías o APIs (*Application Programming Interface*) desarrolladas por Microsoft para facilitar las tareas relacionadas con el multimedia, especialmente videojuegos. Es exclusivo para el sistema operativo Windows, aunque se están desarrollando versiones de código

abierto para sistemas Unix. Entre sus librerías se encuentran *Direct3D* (gráficos) y *DirectInput* (control por teclado/mouse), las cuales utilizaremos en el proyecto.

La versión que se utilizará en la demo es la versión 10 [4], la cual trae toda una serie de mejoras con respecto a versiones anteriores, lo que incluye un lenguaje para *shaders* más avanzado (HLSL 4.0) y la inclusión del *Geometry Shader*, el cual nos será de gran utilidad a la hora de trabajar con ciertas técnicas gráficas procedurales.

2.2.2 Visual Studio 2008

Microsoft Visual Studio es un entorno de desarrollo integrado (o IDE, *Integrated Development Environment*) para sistemas Windows. Soporta una gran cantidad de lenguajes de programación, entre ellos C++, con el que trabajaremos. Se trata de una herramienta muy potente y completa con editor de código, depurador y compilador. Además, permite crear automáticamente el código para gestionar ventanas de Windows de forma rápida y sencilla.

El programa dispone de una licencia de prueba gratuita pero limitada. En nuestro caso, se ha utilizado la licencia de estudiante proporcionada a los que estudiamos en la UAB [5]. La versión utilizada es la *Professional*.

2.2.3 Notepad++

Notepad++ es un editor de código gratuito (aunque puede ser usado como un procesador de textos normal) con licencia GPL (*General Public License*)[2]. La ventaja de este programa, aparte de ser muy flexible, es que contiene ayudas visuales para una gran cantidad de lenguajes, entre ellos C++.

Su uso en este proyecto está dedicado principalmente a editar el código de los *shaders* para nuestra aplicación, ya que este no está soportado directamente por *Visual Studio*.

2.2.4 FX Composer

FX Composer es otro entorno de desarrollo integrado, esta vez dedicado a la edición y depuración de *shaders*. Contiene soporte para DirectX 10, aunque lleva desde 2009 sin ser actualizado[3]. Sin embargo, el uso que le vamos a dar se limita a detectar errores y depurar los *shaders* que creemos, algo que no es posible con *Visual Studio*.

[2] Se puede descargar en http://notepad-plus-plus.org/
[3] Se puede descargar en http://developer.nvidia.com/fx-composer

2.3 Viabilidad técnica

Prácticamente todos los programas y tecnologías utilizados en este proyecto cuentan con una amplia documentación y sencillez de uso, con lo cual el aprendizaje no resulta un problema. Sin embargo, las técnicas procedurales utilizadas para la demo requieren de una extensa investigación y testeo ya que muchos de esos procedimientos son experimentales y no se usan frecuentemente, por lo que la documentación al respecto está bastante dispersa. Esto se reflejará en las horas dedicadas al análisis en la planificación del proyecto.

2.4 Viabilidad legal

No existen problemas de índole legal que comentar en este proyecto. No se utilizan datos de carácter personal y las licencias de todos los programas utilizados son gratuitas o adquiridas legalmente. Lo único que podría haber presentado un problema es la patente de un algoritmo que se usará en la demo, llamado *Marching Cubes*. Sin embargo, dicha patente expiró en 2005, con lo cual se puede usar libremente [6].

2.5 Planificación del proyecto

En esta sección se expondrán las tareas a seguir para desarrollar la demo, divididas en tres partes diferenciadas, con una estimación aproximada del número de horas necesarias para cada tarea:

- Análisis y diseño
- Implementación y optimización
- Documentación

2.5.1 Análisis y diseño

En la fase de análisis se hace el estudio previo sobre el proyecto (lo que incluye el informe previo) y se busca toda la información posible respecto al desarrollo de técnicas procedurales. Dado que la documentación sobre este tema no es muy extensa (más allá de los conceptos más populares, como las texturas procedurales) esta tarea requerirá dedicar un tiempo extra a investigar el tema y hacer pruebas con las diferentes técnicas a utilizar. De esta manera se podrá decidir qué incluir en la demo y qué no. En la tabla 1 podemos ver el recuento de horas.

Estudio de viabilidad	6h
Redacción Informe Previo	4h
Búsqueda de información	20h
Pruebas	10h
TOTAL	**40h**

Tabla 1. Tareas de análisis y diseño

2.5.2 Implementación y optimización

En esta fase se entrará de lleno en el desarrollo de la demo, empezando por la estructura y clases básicas (manejo de la ventana, cámara, input, escribir texto por pantalla,…) para luego implementar cada una de las técnicas que nos interesan una por una.

Una vez se haya conseguido representar todos los efectos necesarios se procederá a intentar optimizar el rendimiento, identificando y solucionando posibles cuellos de botella y corrigiendo errores. En la tabla 2 podemos ver el recuento de horas para cada parte.

Implementación de la aplicación	
Estructura básica	15h
Implementación de efectos	65h
Optimización	
Depuración del código	15h
Pruebas	10h
TOTAL	**105h**

Tabla 2. Tareas de implementación y optimización

2.5.3 Documentación

En la fase de documentación (que se da paralelamente a las otras dos fases) se engloban todas las tareas de redacción de los documentos necesarios para este proyecto, como son la memoria y la presentación. También se incluye la documentación del código mediante comentarios. En la tabla 3 veremos el recuento de horas.

Documentos	
Memoria	25h
Presentación	5h
Código	
Comentarios de las clases	15h
TOTAL	**45h**

Tabla 3. Tareas de documentación

A continuación, se presenta la tabla 4, que incluye las horas totales necesarias.

Análisis y diseño	40h
Implementación	105h
Documentación	45h
TOTAL	**190h**

Tabla 4. Horas totales del proyecto

Capítulo 3

Implementación

En este capítulo se detallará el proceso seguido a la hora de construir nuestra demo con efectos puramente procedurales. Se comentarán también los obstáculos encontrados por el camino, y las posibles soluciones que se ponderaron.

A continuación se presenta un diagrama de módulos simplificado que resume el funcionamiento de nuestra aplicación (figura 3.1).

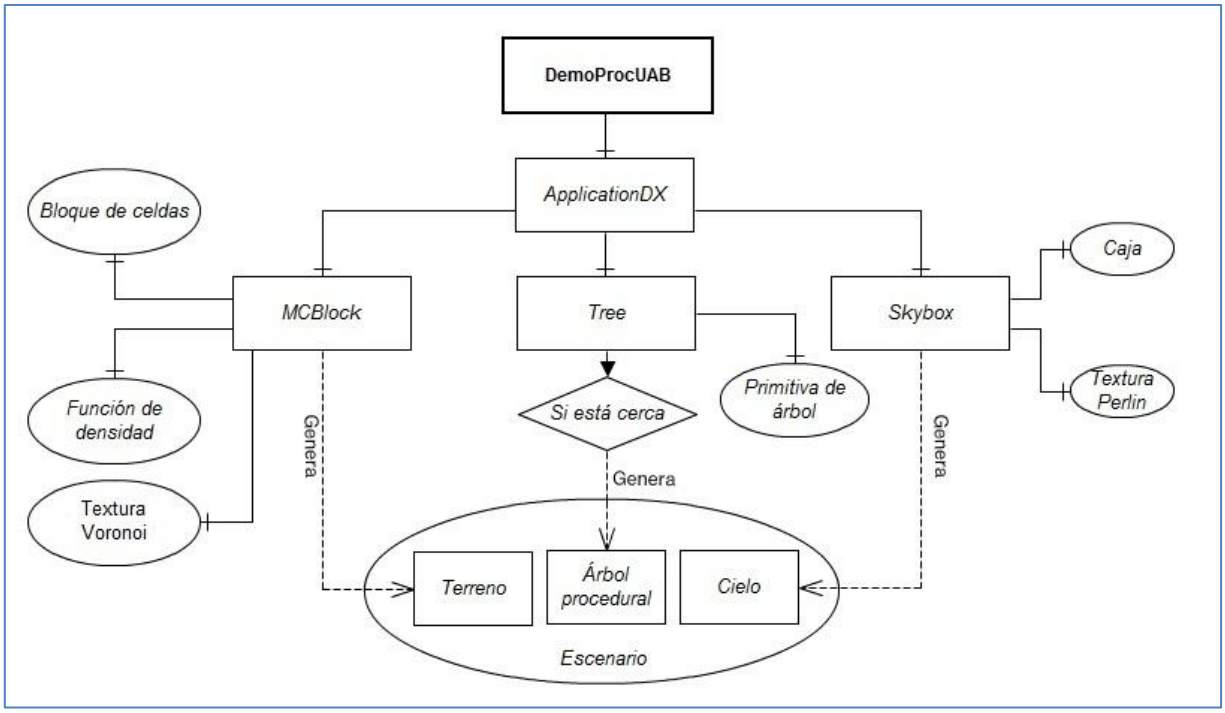

Fig. 3.1 Diagrama de módulos de la aplicación

DemoProcUAB es el punto de entrada a la aplicación. ***ApplicationDX*** gestiona el resto de clases que se dedican a la generación de efectos procedurales. La clase ***MCBlock*** utiliza un bloque de celdas (definido en un *buffer* de vértices) y una función de densidad con la que que se encarga de crear nuestro terreno aleatorio, al cual se aplica una textura generada con ruido Voronoi (anexo A). La clase ***Tree*** gestiona la creación de la vegetación procedural a partir de la primitiva básica de un árbol, siempre que la cámara esté en el área de dibujo. La clase ***Skybox*** genera nuestro cielo a partir

de una simple caja o cubo y una textura procedural con ruido de Perlin (anexo A). Lo que el usuario verá al ejecutar la demo es el conjunto de todos estos elementos en el escenario.

En los siguientes apartados se explicará la implementación de estos elementos punto por punto:

- Generación del terreno aleatorio.
- Vegetación procedural.
- Texturas procedurales.

3.1 Generación del terreno

El aspecto más interesante de los terrenos procedurales es que podemos añadirles un factor de aleatoriedad. De esta forma obtendremos un mundo nuevo cada vez que ejecutemos nuestros procedimientos, una cualidad de mucho valor en un videojuego, por ejemplo. Ese es el acercamiento que vamos a seguir para generar nuestro terreno, usando los algoritmos descritos en el anexo A.

Dado que dicho terreno va a ser creado y actualizado en tiempo real, lo que haremos es crear un bloque singular que generará el terreno alrededor nuestro, allá donde nos movamos. En el siguiente apartado se detallará este proceso en tres partes: la **implementación y visualización** del terreno, el cálculo de la **iluminación** y la creación de una **función de densidad** interesante.

3.1.1 Implementación y visualización

Utilizaremos el algoritmo *Marching Tethraedrons* para generar nuestra malla poligonal a partir de una función de densidad. Primero de todo tendremos que definir nuestra estructura de datos y a continuación la manera de ejecutar el algoritmo de forma eficiente.

Definiremos nuestro bloque como una cuadrícula tridimensional de celdas, cada una de las cuales estará dividida en los 6 tetraedros que componen dicha celda. Para ello, crearemos un *vertex buffer*[4] con la información de los vértices que componen el bloque, junto con su *index buffer*[5] correspondiente. El tamaño del bloque puede variarse según nuestras necesidades. En nuestro caso, el tamaño asignado por defecto es de 48x48x24 celdas (aunque realmente el bloque mide 49x49x25 si contamos las esquinas de cada celda).

A continuación, podríamos definir una función de densidad y aplicarla celda por celda para generar nuestros polígonos en tiempo de CPU. Sin embargo, hacerlo así resultaría ineficiente ya que deberemos iterar sobre una gran cantidad de celdas. Por tanto, lo más adecuado es delegar ese trabajo

[4] Estructura de datos que contiene información de los vértices que se van a dibujar en pantalla.
[5] Contiene los índices que indican en qué orden dibujar los vértices en el *vertex buffer*

a nuestra GPU. Es aquí donde entra en juego el *Geometry Shader* de DirectX 10, ya que nos permitirá generar la geometría de la superficie al mismo tiempo que evaluamos la función de densidad en nuestro bloque de celdas, todo en una sola llamada a la rutina de dibujo [12]. El proceso a seguir es el siguiente:

1. Pasamos el *vertex buffer* que define nuestro bloque de celdas a la GPU para dibujarlo en pantalla (aunque realmente no vamos a hacerlo). La única información que le pasaremos es la posición de los vértices.
2. Dentro del *Vertex Shader* aplicaremos las transformaciones de coordenadas habituales a los vértices, guardándonos adicionalmente la posición en coordenadas mundo. Utilizaremos dicha coordenada para evaluar la función de densidad y guardarnos el valor devuelto en nuestra estructura, la cual pasaremos al *Geometry Shader*.
3. Obtendremos grupos de 4 vértices (los que forman el tetraedro) y aplicaremos nuestro algoritmo para encontrar intersecciones de la superficie. En caso positivo, se generan hasta 2 triángulos por tetraedro.
4. Una vez en el *Pixel Shader*, calculamos el color a mostrar y finalizamos el proceso de dibujado.

Hay una razón por la que utilizamos el algoritmo *Marching Tetrahedrons* (MT) en vez del clásico *Marching Cubes* (MC). El rendimiento del *Geometry Shader* viene determinado por la cantidad de datos que tiene que generar como salida al buffer *StreamOut*. Si utilizáramos MC, deberíamos coger hasta 8 vértices de entrada y generar hasta 5 posibles triángulos por cubo, lo cual implica una salida de hasta 15 vértices, cada uno con información de posición y la coordenada mundo (necesaria para calcular normales), lo cual equivale a un máximo de 105 números en punto flotante saturando el ancho de banda. Usando MT, solo tenemos que lidiar con 4 vértices de entrada, y un máximo de 4 vértices de salida (utilizando una topología de líneas adyacentes), equivalente a 28 números en punto flotante. Aunque el usar tetraedros aumente el número de iteraciones, mejora enormemente el rendimiento del GS al reducir nuestro ancho de banda.

Si programamos las matrices de traslación para que el bloque se mueva al mismo tiempo que la cámara, obtendremos un bloque que genera polígonos allá donde nos movamos, precisamente lo que andábamos buscando. En la figura 3.2 se puede ver dicho bloque en malla de alambre con una función de densidad que genera un plano.

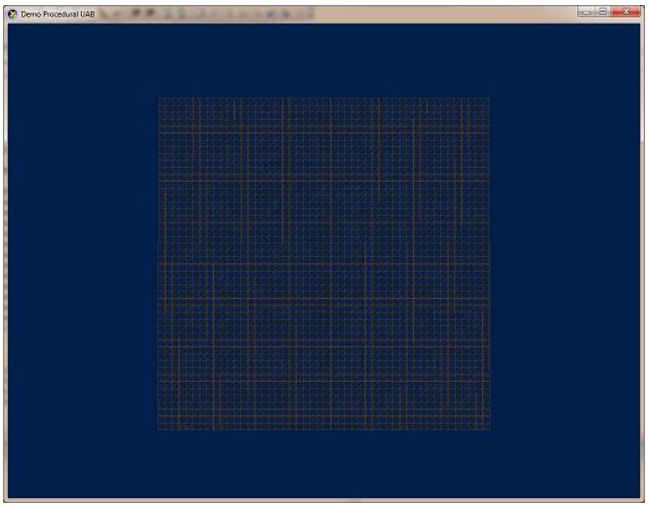

Fig. 3.2 Bloque de terreno visto desde arriba. Función Density = –ws.y

3.1.2 Iluminación

Ahora que ya tenemos nuestra malla auto-generada, podríamos darle algo de forma modificando nuestra función de densidad.

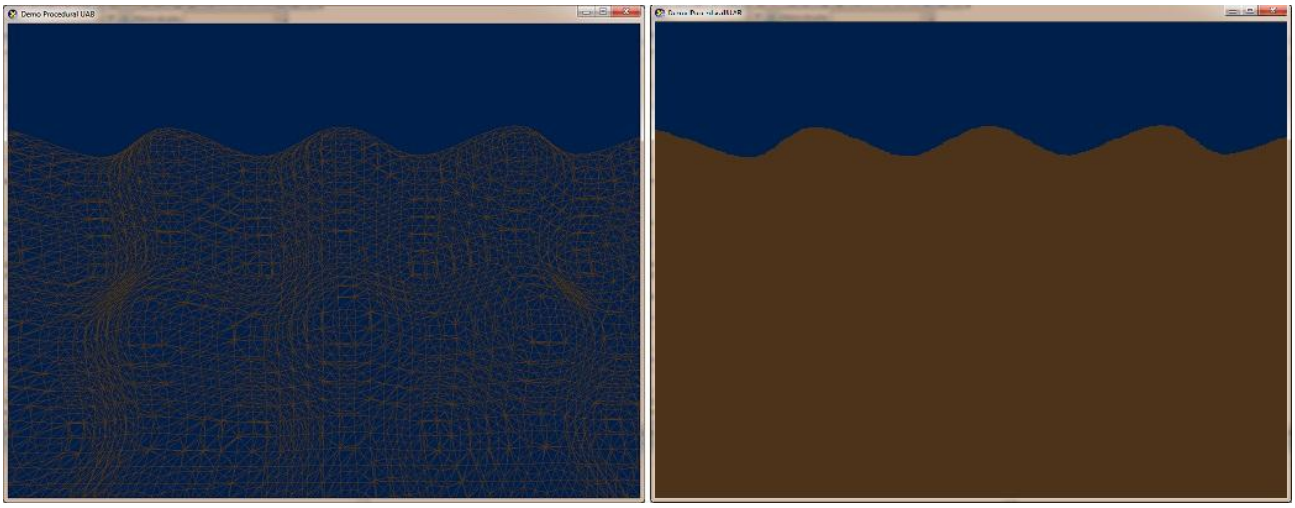

Fig. 3.3 A la izquierda, función sinusoidal en alambre, a la derecha en color sólido

Como podemos ver en la figura 3.3, el hecho de devolver únicamente color sólido no nos permite apreciar correctamente la forma de nuestro terreno. Para solventar ese problema, debemos utilizar algún sistema de iluminación y aplicarlo a la malla. Aquí nos topamos con un problema flagrante: para calcular luz necesitamos los vectores normales de los polígonos generados. Dado que estamos creando nuestra geometría sobre la marcha, no disponemos de esa información de antemano, por lo tanto debemos calcular dichos vectores a la vez que generamos polígonos.

Una manera de hacerlo sería calcular vectores normales a las caras haciendo un producto vectorial entre las aristas del polígono, pero con eso solo conseguiríamos una iluminación plana. Para

conseguir una iluminación suave, debemos obtener las normales de los vértices. Para ello necesitaríamos obtener las normales de las caras adyacentes y calcular el promedio, pero tal y como está implementado nuestro algoritmo en el GS (generando uno o dos polígonos por tetraedro como mucho) no es posible obtener esa información sin tener que hacer múltiples iteraciones al *shader*.

Sin embargo, tenemos una fórmula de densidad que define un campo escalar. Por lo tanto, lo que podemos hacer es calcular el gradiente de dicho campo en el punto (o vértice) seleccionado. Para ello, se evaluará la función de densidad dos veces por coordenada:

$d = distancia\ entre\ celdas \mid ws = coordenada\ mundo \mid grad = gradiente$

$grad.x = Density(ws + vector[d, 0,0]) - Density(ws + vector[-d, 0,0])$

$grad.y = Density(ws + vector[0, d, 0]) - Density(ws + vector[0, -d, 0])$

$grad.z = Density(ws + vector[0,0, d]) - Density(ws + vector[0,0, -d])$

Efectuando esta operación en el *Pixel Shader*, obtendremos un gradiente aproximado que equivale a las normales de los vértices. Con esa información, podemos calcular fácilmente un modelo de iluminación tipo *Blinn-Phong* [13]. En la figura 3.4 podemos ver el efecto ya aplicado.

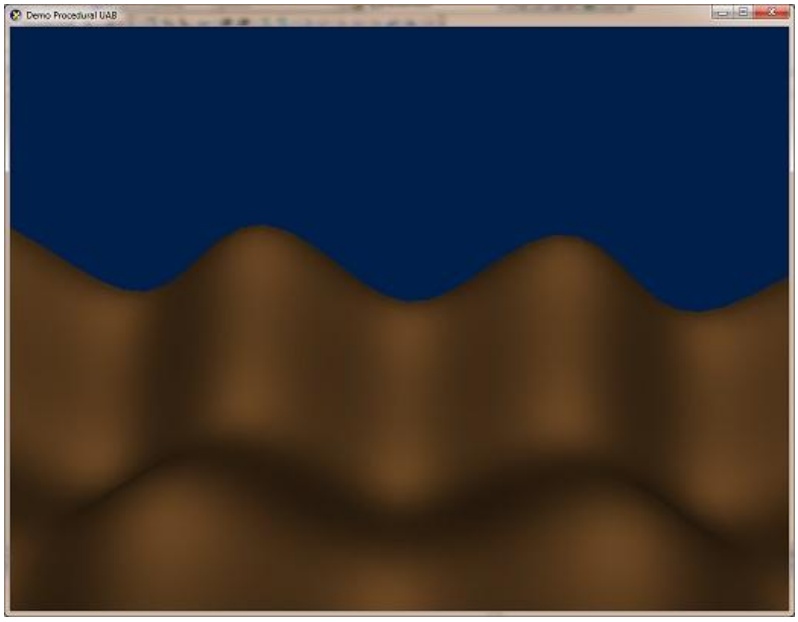

Fig. 3.4 Función sinusoidal con iluminación aplicada

3.1.3 Función de densidad

Ahora que ya tenemos un bloque auto-generado funcional y un sistema de iluminación establecido, somos libres de modificar nuestra función de densidad para obtener un terreno más interesante, además de añadir el factor de aleatoriedad.

Ya hemos visto anteriormente imágenes del terreno representando un “suelo” y una función sinusoidal. Esta vez vamos a añadir algo de ruido aleatorio. Para ello, utilizaremos el ruido Simplex en 3D, implementado en el *shader*.

Primero de todo, variaremos ligeramente la frecuencia y amplitud de nuestra función sinusoidal para aliviar la repetitividad del patrón (figura 3.5):

$$density = -ws.y + 1.6 * sin(-0.53 * ws.x) + 0.82 * sin(-0.66 * ws.z)$$

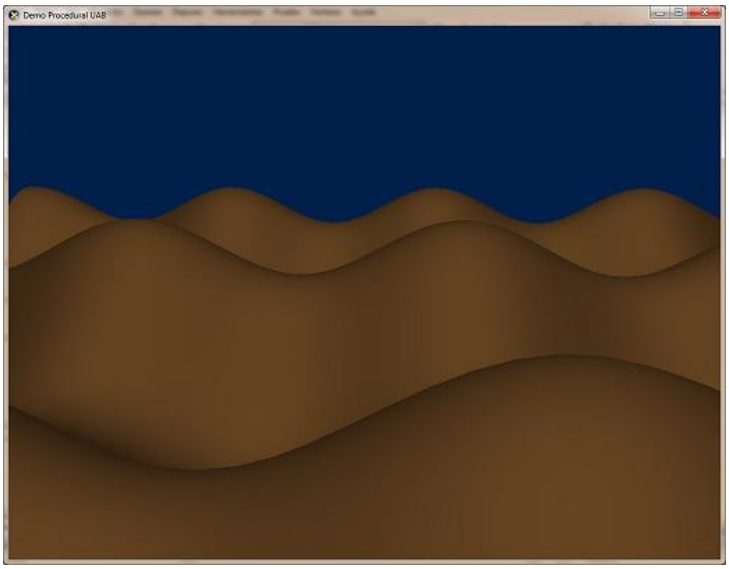

Fig. 3.5 Función sinusoidal alterada

Si bien el patrón ha cambiado, no se observa una gran diferencia. Es hora de añadir algo de ruido a la ecuación. Vamos a probar añadiendo una octava de ruido muestreado a baja frecuencia y amplitud (figura 3.6).

$$density += (SimplexNoise3D(ws * 0.33).x * 0.64)$$

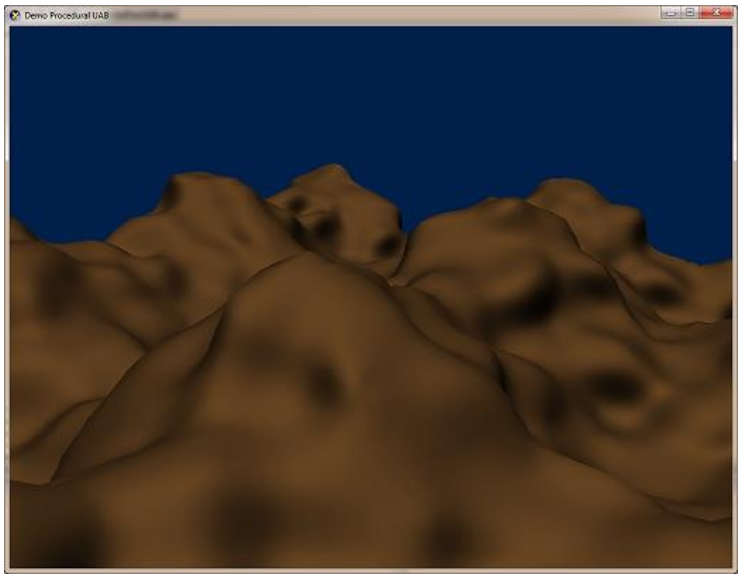

Fig. 3.6 Función sinusoidal con una octava de ruido

Esto mejora el resultado, pero se puede ampliar. Una sola octava de ruido no ofrece mucha variedad, pero podemos sumar octavas de ruido adicionales para añadir más detalle, generando así un **ruido fractal**. Si añadimos octavas a frecuencias menguantes y amplitudes cada vez más grandes, se formarán cuerpos grandes como montañas. Si lo hacemos al revés, añadiremos más detalle a corta distancia. Probaremos el primer método, añadiendo de 3 a 5 octavas (figuras 3.7 y 3.8):

$$density += (SimplexNoise3D(ws * 0.115).x * 1.05)$$

$$density += (SimplexNoise3D(ws * 0.063).x * 2.11)$$

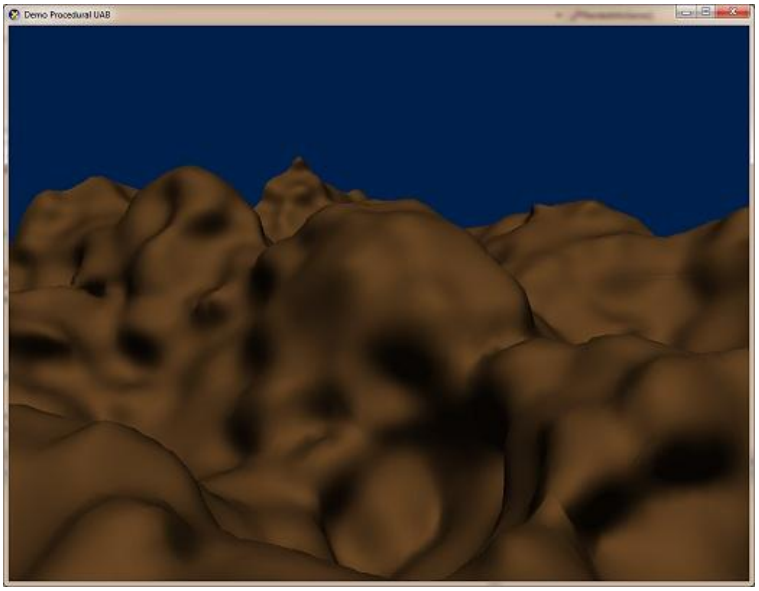

Fig. 3.7 Función sinusoidal con 3 octavas de ruido

$$density += (SimplexNoise3D(ws * 1.18).x * 0.165)$$

$$density += (SimplexNoise3D(ws * 0.67).x * 0.31)$$

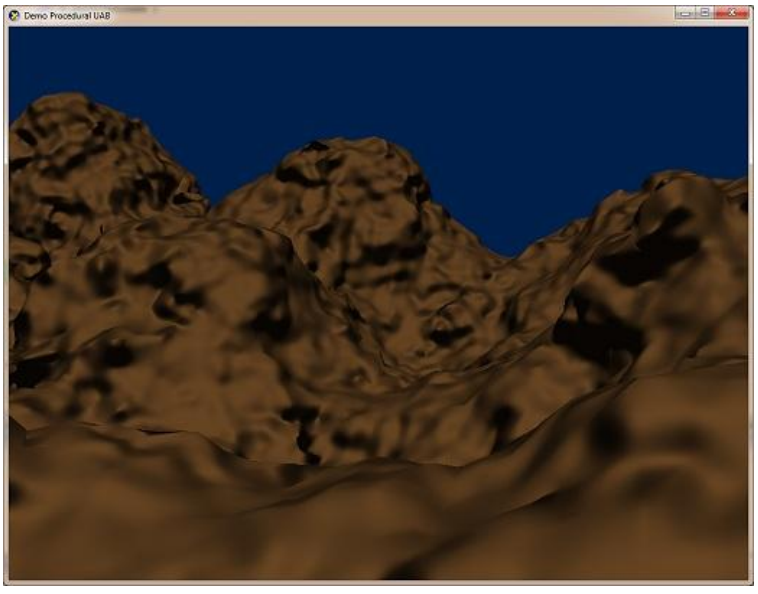

Fig. 3.8 Función sinusoidal con 5 octavas de ruido

El problema de añadir octavas de ruido es que estamos calculando dicha función a cada *frame*, debido a que generamos el terreno en tiempo real. La función de ruido es costosa, y teniendo en cuenta que al calcular las normales aplicamos dicha función seis veces más, incurrimos en un cuello de botella. Sin embargo, usando 3 o 4 octavas de ruido ya se obtiene un paisaje interesante con un rendimiento relativamente estable, por lo que no necesitamos añadir más.

Opcionalmente, podemos añadir otros efectos menos costosos de calcular, como el *pre-warping* (añadir una perturbación a la coordenada mundo antes de aplicar el ruido) o un suelo (sustrayendo la altura deseada a la coordenada mundo). Podemos ver un ejemplo en la figura 3.9.

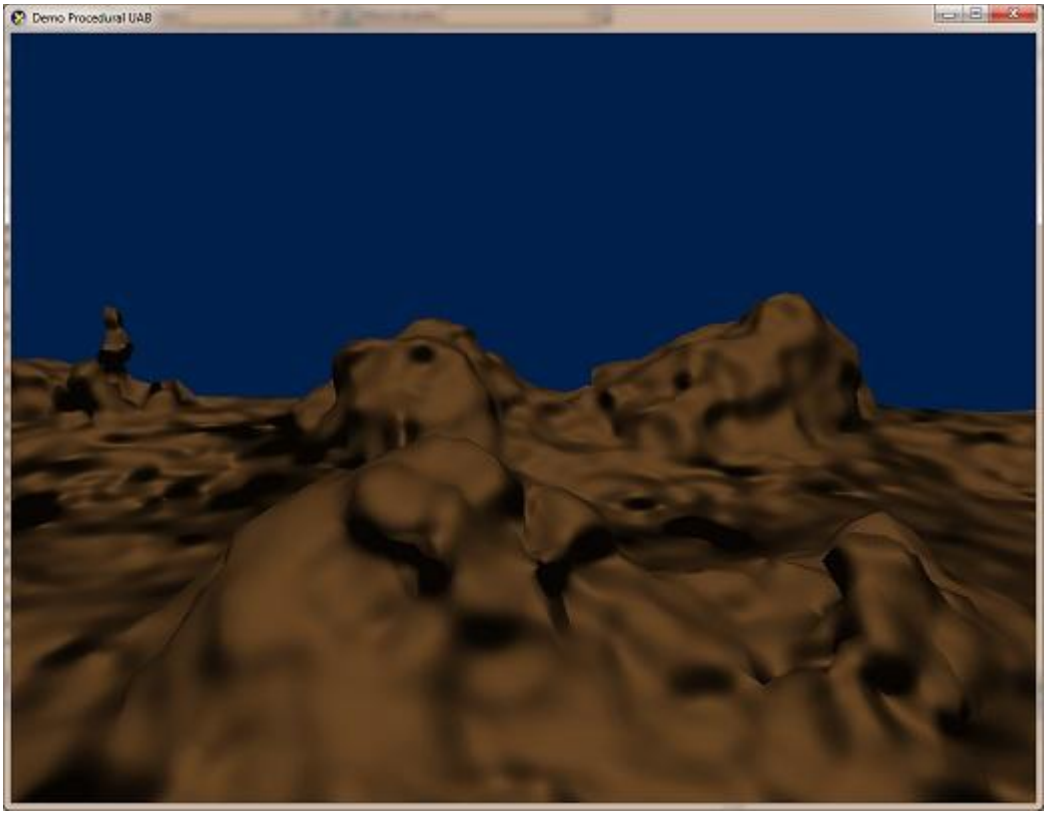

Fig. 3.9 Terreno con pre-warping y 4 octavas de ruido

3.2 Vegetación procedural

Una vez creado nuestro terreno aleatorio, nos interesa el poder añadir algún elemento más a nuestro escenario. La vegetación procedural resulta interesante por la amplia variedad de flora que se puede recrear. No obstante, la generación de plantas es un tema ya ampliamente estudiado y utilizado. Hoy

en día la mayoría de entornos "verdes" (bosques, junglas, etc) son generados con técnicas procedurales usando programas *middleware* muy conocidos, como *SpeedTree*[6].

Dada la complejidad de este tema, vamos a centrarnos en desarrollar una versión mucho más simple de estas técnicas: generaremos un árbol a partir de una primitiva básica, la cual se expandirá con el uso del *Geometry Shader*. Se explicará primero el **proceso de construcción del árbol**, y seguidamente el **método para posicionarlo** en nuestro terreno aleatorio.

3.2.1 Construcción del árbol

Como primitiva utilizaremos un modelo creado vértice por vértice con la posición y normales ya definidas, pero al que añadiremos un parámetro adicional que llamamos "grow". Dicho parámetro indicará si el vértice en cuestión debe ser extruido o no (y con ello todo el polígono) y el tamaño de dicha extrusión. Para marcar la dirección de extrusión utilizaremos la normal del vértice, la cual perturbaremos ligeramente con un factor aleatorio, para cambiar el aspecto del árbol en cada ejecución (diagrama explicativo en la figura 3.10).

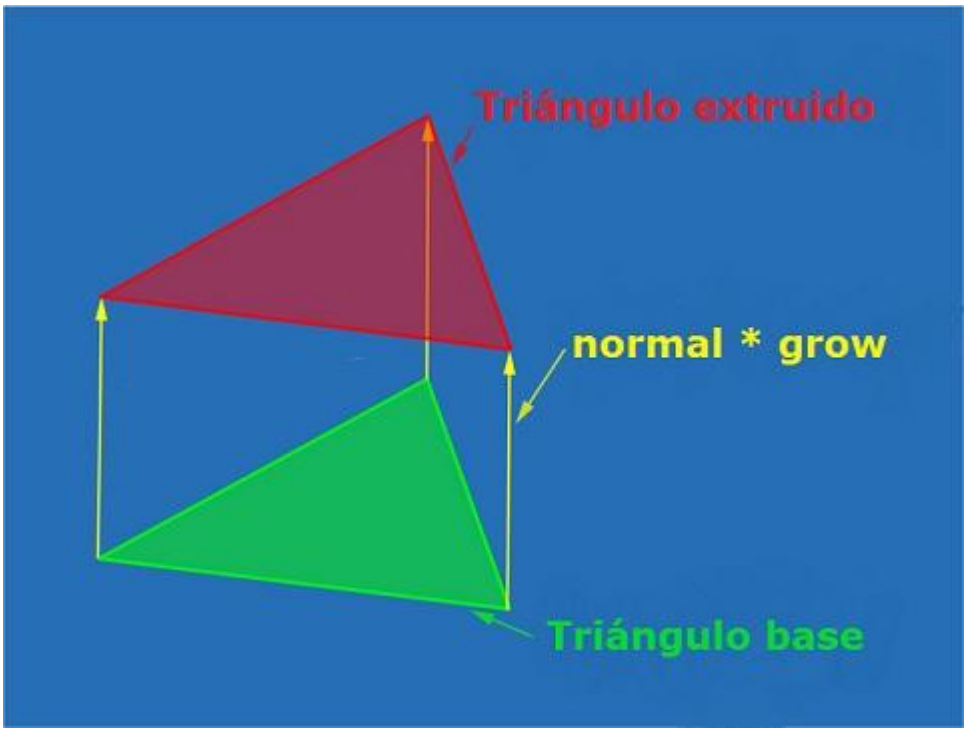

Fig. 3.10 Diagrama explicativo de la extrusión de caras

El proceso es el siguiente:

[6] Ver http://www.speedtree.com/

1. Cargamos nuestra primitiva en un *vertex buffer* básico y la dibujamos por pantalla en una pasada.
2. Copiamos la información del modelo actual a un *buffer* adicional al que llamaremos "DrawFrom", desde el que dibujaremos. Hacemos una pasada por el GS y hacemos la extrusión de las caras "marcadas" por el valor "grow", añadiendo modificadores para variar la longitud, dirección y probabilidad de generar más ramas.
3. Una vez creado el modelo "expandido" se pasa la información al *StreamOut buffer*, el cual realimentará al GS para una nueva expansión sobre el modelo ya expandido. Esta información se guardará en el *buffer* "StreamTo".
4. Se hace un *swap* entre el *buffer* "StreamTo" con el modelo expandido y "DrawFrom"
5. Repetimos desde el paso 2 hasta que lleguemos al máximo de vértices permitidos o se decida según los parámetros que no hay que expandir más las ramas.

Para expandir las ramas, se utilizan 4 modificadores distintos que definen la longitud, dispersión, acortamiento (cuanto reducimos el siguiente nivel de ramas) y el *ratio* de extinción (valor con el que decidimos si hay extrusión o no). Para decidir la dirección de expansión se utiliza un vector de números pseudo-aleatorios indexado por la ID del vértice, una variable de sistema que se añade automáticamente en el *shader* y viene con cada vértice. En la figura 3.11 podemos ver el resultado.

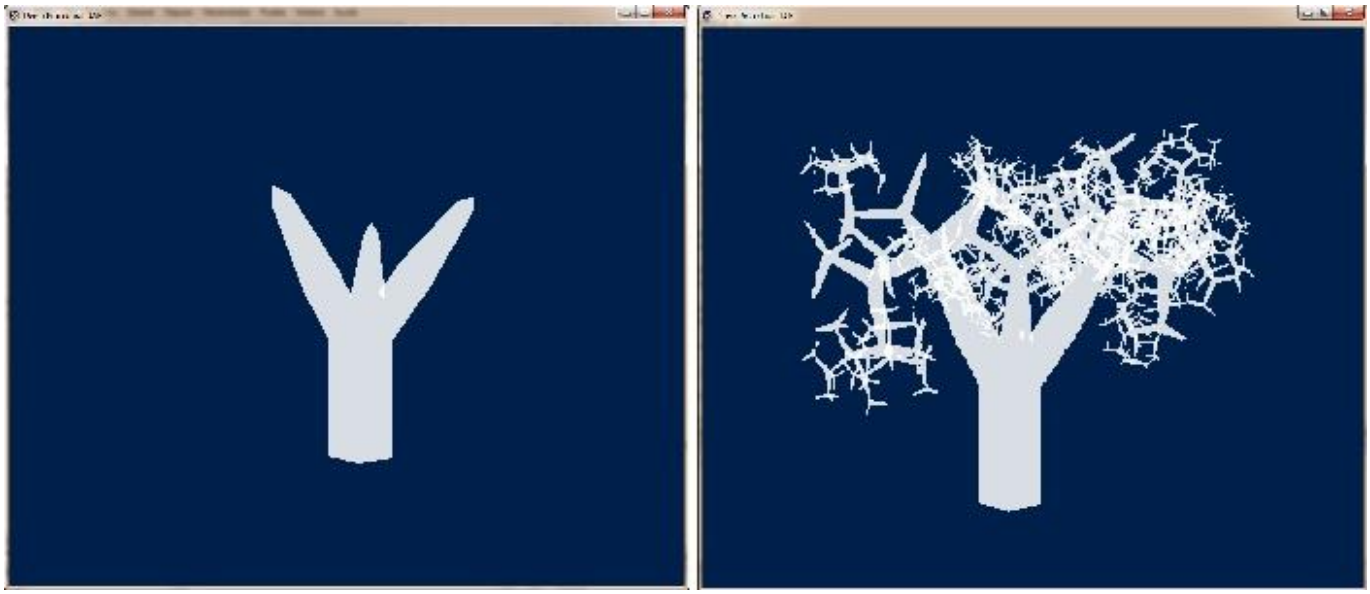

Fig. 3.11 A la izquierda, la primitiva. A la derecha, un modelo expandido

3.2.2 Posicionamiento

Una vez creado el árbol, nos interesa poder colocarlo en algún lugar de nuestro terreno. Sin embargo, esto presenta un problema: la generación de terreno es totalmente aleatoria, con lo que no podemos saber con exactitud de una coordenada donde el árbol quedara "fijo" en la tierra.

Podemos hacer, no obstante, un pequeño truco: modificamos la función de densidad del terreno para que en un punto concreto que definamos se forme una planicie redonda a la altura deseada. De esta forma, independientemente de la forma que adquiera nuestro escenario, sabremos que hay un punto donde colocar el árbol. Cierto es que no siempre se formará correctamente la planicie, ya que el factor aleatorio influye en la forma que se coge. En la figura 3.12 podemos ver el resultado.

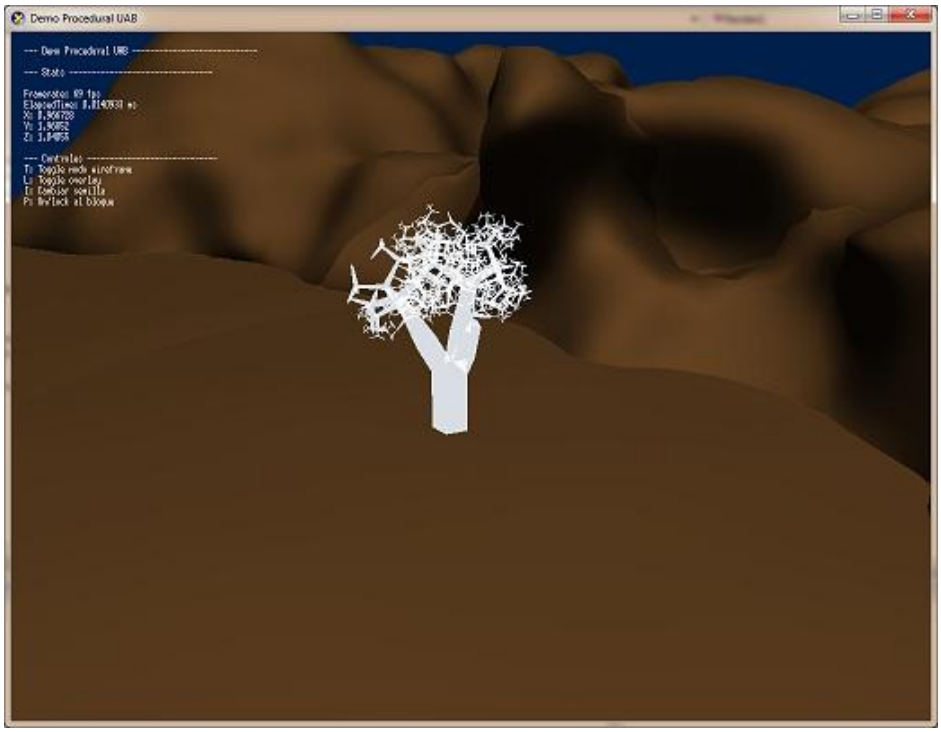

Fig. 3.12 El árbol posicionado en el terreno

3.3 Texturas

Al igual que con la vegetación procedural, el campo de las texturas procedurales ha sido ampliamente estudiado y actualmente se trabaja intensivamente con él. Programas como *Photoshop* o *3D Studio* tienen sus propios módulos para generar efectos procedurales en texturas. Pero teniendo en cuenta nuestra norma de "no cargar nada desde archivos", lo que haremos en la demo será generar esas texturas en tiempo de ejecución, con lo que puede ser interesante ver sus efectos así como sus

pros y contras. A continuación se explicará la implementación de una textura de **cielo** y otra para el **terreno**.

3.3.1 Cielo

Para generar una textura de cielo que envuelva nuestro escenario, primeramente construimos un sencillo *skybox*, esto es, una caja gigante donde pintaremos la textura por sus caras interiores.

A continuación, haremos uso del ruido Simplex 3D para generar ruido fractal que simule nubes. Además, utilizaremos diferentes parámetros para determinar la cantidad de nubes en pantalla y su tamaño.

Dados:

Position: coordenada mundo de un punto de la caja.

Scale: controla el tamaño del patrón de nubes. Rango [0-∞]

Clearness: determina lo despejado que estará el cielo. Rango [0-1].

RuidoPerlin(w,n): función de ruido usando punto *w* y sumando *n* octavas.

Lerp(a,b,desp): hace una interpolación entre *a* y *b* usando *desp* como intervalo.

Calculamos:

$$Position = Position * Scale$$
$$Noise = RuidoPerlin(Position, 4)$$
$$Overcast = (1 - Clearness)^4$$
$$Cloudcolor = lerp(color\ blanco, color\ gris\ oscuro, 1 - Overcast)$$
$$Skycolor = lerp(color\ azul, Cloudcolor, Noise)$$

Siendo *Skycolor* el valor final que se calcula para el color de cada píxel del cielo.

Lo más interesante de usar el ruido en tres dimensiones es que la textura no tiene que adaptarse a las coordenadas UV del cubo; de hecho, las ignora, ya que solo tiene en cuenta la posición de los vértices para calcular el valor. Además, podemos obtener la variable de tiempo transcurrido de la aplicación para generar una textura animada: simplemente desplazamos la coordenada usada para muestrear el ruido en el eje X, consiguiendo que las nubes se muevan.

Podemos ver una captura del resultado en la figura 3.13. No es un efecto especialmente vistoso, pero cumple su cometido.

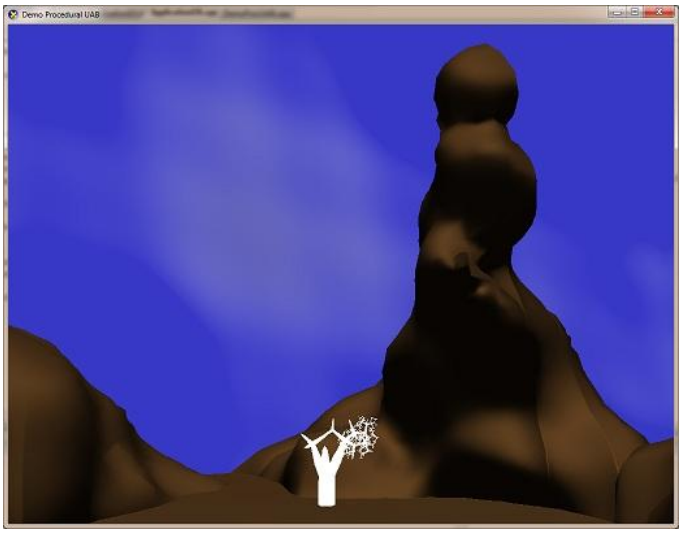

Fig. 3.13 Captura del cielo con nubes

3.3.2 Terreno

En este caso utilizaremos el ruido de Voronoi (Anexo A) para generar la textura para el terreno. Resulta todavía más sencillo que con el ruido Simplex, ya que basta con calcular el ruido y aplicar la textura en cada punto usando el *Pixel Shader* para obtener un resultado bastante satisfactorio. Puede que el aspecto membranoso de Voronoi no case especialmente con lo que sería una textura de tierra, pero como ya se comentó en el capítulo 2, la búsqueda de realismo no es una prioridad.

Al igual que con el ruido de Simplex, el ruido de Voronoi se genera usando coordenadas de 3 dimensiones, por lo que es perfecto para "arropar" a nuestro terreno generado arbitrariamente. Además, podemos meter en juego la variable de tiempo para crear un efecto de movimiento en las celdas, dándole un aspecto más gelatinoso si cabe. En la figura 3.14 podemos ver el resultado.

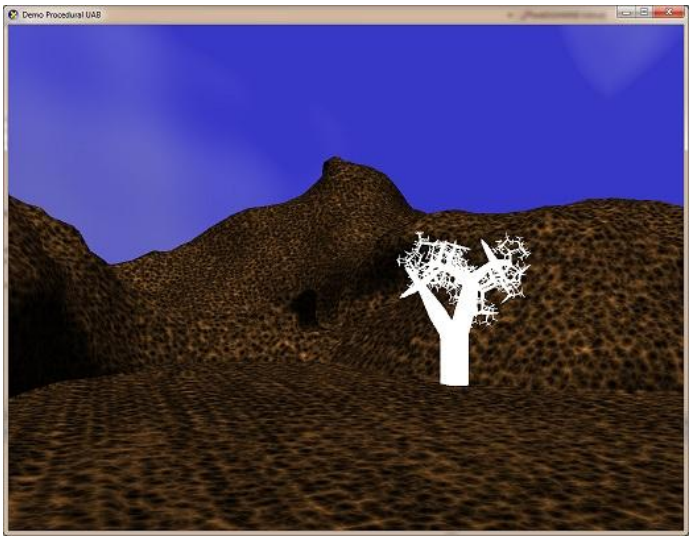

Fig. 3.14 Terreno texturizado con ruido de Voronoi

Capítulo 4

Resultados

En este capítulo se evalúan los resultados obtenidos de la demo finalizada, se discuten las posibles mejoras que se podrían haber implementado y la viabilidad que tendrían las técnicas procedurales implementadas en proyectos más grandes como un videojuego. Se tratan los siguientes puntos:

- Pruebas de rendimiento.
- Discusión de resultados.

4.1 Pruebas de rendimiento

De entre las pruebas realizadas, se han evaluado principalmente los FPS[7], activando uno o varios de los efectos creados, con tal de hacer una media de los FPS que obtenemos en cada caso. De esta forma podemos dilucidar qué efecto es más costoso para el rendimiento. A continuación se presenta una tabla con los valores obtenidos. El tamaño del bloque de terreno utilizado es el estándar de 48*48*32 y el PC utilizado es un *Intel* I7 950 con una tarjeta gráfica *GeForce* GTX 460.

La X indica que el efecto ha sido activado.

Terreno	Textura Voronoi	Cielo	Árbol	FPS
				1600-1700
X				100-150
X	X			90-100
		X		~425
			X	450-600
		X	X	200-350
X	X	X		75-100
X	X		X	66-90
X	X	X	X	65-85

Tabla 5. Medida de FPS según los efectos activados

[7] *Frames* por segundo

4.2 Discusión de resultados

Si bien los valores arriba expuestos parecen ser positivos (todos por encima de 60 *frames*) no podemos olvidar que la demo ha sido ejecutada en una máquina relativamente potente. En el hipotético caso de un PC de gama media, los resultados podrían no ser tan satisfactorios.

El elemento más costoso para el rendimiento es el terreno aleatorio, como podemos ver. Esto está ligado con el cuello de botella ya mencionado en el capítulo 4 respecto al cálculo de la iluminación y desgraciadamente es algo que no se puede solventar con facilidad.

Dejando a un lado el tema del rendimiento, podemos decir que la demo cumple con los objetivos que nos habíamos marcado en un principio. Creada al 100% con efectos procedurales, con un escenario generado aleatoriamente (el cual podemos cambiar en tiempo de ejecución) y que podemos explorar libremente. Además, podemos apreciar otros efectos procedurales como las texturas del cielo y el terreno, así como un árbol generado proceduralmente.

Dentro de la demo las acciones posibles son: mover la cámara por el escenario, cambiar la semilla del generador de números aleatorios para modificar el terreno, hacer que el bloque se mueva con nosotros o no, activar/desactivar el *overlay* de texto y activar/desactivar el modo alambre. Más que suficiente para lo que se pretendía.

Es importante decir que la norma de “no cargar nada desde archivo” ha acabado teniendo un impacto negativo en el desarrollo. Por ejemplo, los problemas de rendimiento con el terreno se podrían haber solventado cargando archivos de volumen con valores de ruido ya incorporados, con lo que solo hubiera bastado con muestrearlos. Además el uso de texturas procedurales en tiempo real, si bien conveniente para aplicar en objetos geométricamente arbitrarios y complejos, deberían ser relegadas a casos muy concretos a favor del uso de texturas cargadas desde archivos, con tal de no tener que añadir cálculos de ruido adicionales al ciclo de dibujado. En definitiva, se puede decir que un modelo 100% procedural no es precisamente efectivo.

Capítulo 5

Conclusiones

- En este proyecto se ha creado una demo 3D funcional que genera un terreno aleatorio explorable, vegetación procedural y utiliza texturas procedurales.
- Se ha experimentado con una serie de técnicas no muy prolíficas en el sector de los gráficos por computador. Para ello, se ha hecho uso de tecnologías relativamente modernas como *DirectX 10*.
- Se ha analizado la viabilidad del proyecto y se han establecido una serie de objetivos alcanzables.
- Se ha desarrollado una aplicación 3D que cumple con nuestras premisas iniciales y que puede obtener un rendimiento decente en máquinas modernas.
- Se ha documentado el proyecto convenientemente (tanto memoria como código fuente) en vías de que pueda ser útil a futuros estudiantes de la materia.

Este proyecto no ha estado exento de problemas. Ha sido un desarrollo largo y complicado, afectado por una planificación inicial demasiado ambiciosa y la falta de experiencia en la programación de gráficos por computador. A esto también contribuye que el tema elegido puede llegar a ser muy amplio y compuesto de diferentes especialidades que, por sí solas, podrían abarcar un proyecto entero.

A continuación se detallan algunas posibles mejoras para la aplicación. Serían las siguientes:

- **Generar diferentes bloques de terreno.** En vez de generar el terreno a cada paso, se cargaría bloque a bloque para permitir una mayor distancia de visión y evitar el efecto "onda" de la implementación actual.
- **Crear un modelo híbrido entre técnicas procedurales y tradicionales.** Sería lo más efectivo en caso de querer hacer una aplicación totalmente interactiva.

- **Implementar colisiones y física.** Sería interesante ver cómo reacciona un objeto con física en un terreno generado aleatoriamente.
- **En definitiva, desarrollar un minijuego con la tecnología usada.** De esta manera podríamos poner en prueba las técnicas implementadas para un entorno totalmente interactivo.

Existe un gran futuro para este tipo de técnicas. Motores gráficos modernos como el *Cryengine*[8] ya incluyen generación de terreno por vóxeles y muchos desarrolladores independientes están experimentando con esta tecnología (ya se ha mencionado el caso de *Minecraft*). El futuro pasa por fusionar la generación de gráficos procedurales con el sistema tradicional para crear un modelo eficiente que aproveche las virtudes de cada tipo de técnicas.

Como conclusiones finales y personales, debo comentar que pese a los problemas iniciales he disfrutado aprendiendo sobre este tema, ya que tiene un potencial enorme por explotar. Además, me ha servido para profundizar en tecnologías como *DirectX* y el desarrollo de gráficos por computador.

Espero y deseo que este proyecto pueda animar a otra gente a investigar sobre este fascinante tema. Si bien no es un camino de rosas, la recompensa merece la pena.

[8] Ver http://www.crytek.com/cryengine

Bibliografía

[1] http://pcg.wikidot.com/. Procedural Content Generation Wiki. Web dedicada a la generación procedural en videojuegos. (Última consulta en Junio de 2012)

[2] http://escena.org/wiki/page/Inicio/. Wiki | escena.org. Wiki de la página dedicada a la *demoscene* en español. (Última consulta en Junio de 2012)

[3] http://c.conclase.net/curso/index.php. Curso de C++. Extenso tutorial on-line sobre el lenguaje C++. (Última consulta en Junio de 2012)

[4] http://msdn.microsoft.com/en-us/library/bb205067. *Programming guide for Direct3D 10*. Guía de ayuda sobre DirectX, proporcionada por Microsoft. (Última consulta en Junio de 2012)

[5] https://msdnaa.uab.es/msdnaa/validacio.php/. MSDN Academic Alliance – Universitat Autònoma de Barcelona. (Última consulta en Junio de 2012)

[6] http://www.google.com/patents?vid=4710876. *Patent US4710876*. Descripción de la antigua patente para *Marching Cubes*. (Última consulta en Junio de 2012)

[7] Frank D. Luna. *Introduction to 3D Game Programming with DirectX 10*. Wordware Publishing, 2008.

[8] http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. *Simplex noise Demystified*. Documento explicativo del ruido de Perlin y el ruido Símplex. (Última consulta en Junio de 2012)

[9] http://aftbit.com/cell-noise-2/. *An In-Depth Cell Noise Tutorial*. Explicación del ruido de Voronoi, completo con código de ejemplo. (Última consulta en Junio de 2012)

[10] David S. Ebert. *Texturing and modeling, Second Edition: A Procedural Approach*. Academic Press, 1998.

[11] http://paulbourke.net/geometry/polygonise/index.html. *Polygonising a scalar field (Marching Cubes)*. Documento escrito por Paul Bourke que detalla el funcionamiento de los algoritmos *Marching Cubes* y *Marching Tetrahedrons*. (Última consulta en Junio de 2012)

[12] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html *GPU Gems 3 - Chapter 1 . Generating complex procedural terrains using the GPU*. Se detalla un método avanzado para generar terrenos procedurales utilizando *shaders*. (Última consulta en Junio de 2012)

[13] http://takinginitiative.net/2010/08/30/directx-10-tutorial-8-lighting-theory-and-hlsl/ *Lightning theory and HLSL*. Tutorial para programar modelos de iluminación relativamente sencillos en *shaders*. (Última consulta en Junio de 2012)

[14] William E. Lorensen, Harvey E. Cline: *Marching Cubes: A high resolution 3D surface construction algorithm*. Computer Graphics, Vol. 21, Nr. 4, Julio de 1987

ANEXOS

Anexo A

Conceptos teóricos

En este anexo se expondrán algunos conceptos y algoritmos que serán usados a lo largo del proyecto con el objetivo de entender mejor el proceso de implementación del mismo. Veremos los siguientes conceptos:

- La *pipeline* gráfica
- La generación de ruido, lo cual incluye Simplex y Voronoi.
- La representación gráfica de volúmenes con los algoritmos *Marching Cubes* y *Marching Tetrahedrons*.

A.1 La pipeline gráfica

Dado que en esta demo se va a hacer un uso intensivo de *shaders* ejecutados desde la GPU resulta conveniente explicar el funcionamiento interno del sistema, de manera que sea más sencillo entender las acciones seguidas en el capítulo 3.

A la hora de representar una escena 3D, es necesario pasar a nuestra GPU la información de los vértices de cada objeto (posición en el espacio, coordenadas de textura, vector normal, etc), además de otros datos opcionales como los índices (que apuntan el orden a dibujar dichos vértices, acelerando el proceso). Una vez efectuada la llamada a la rutina para dibujar dichos vértices, todos los datos pasan por lo que llamamos la *pipeline* gráfica, o "tubería gráfica" [7], compuesta de diferentes fases, las cuales aplican los cálculos y transformaciones necesarias para representar los objetos dibujados en pantalla, aplicando por el camino los efectos visuales que hayamos programado en los *shaders*.

A continuación se resume brevemente el funcionamiento de cada fase:

- **Input Assembler (IA):** se encarga de leer la información geométrica de los *buffers* rellenados por el usuario y convertirlo en primitivas geométricas básicas (el tipo de primitiva se define previamente por el programador).

- **Vertex Shader (VS):** coge como entrada los vértices introducidos y les aplica los cálculos que le definamos. Típicamente se usa para transformar las coordenadas de los vértices, pero tiene más usos, como veremos.
- **Geometry Shader (GS):** coge como entrada primitivas enteras (podemos definir el número de vértices de entrada y el máximo de salida). Tiene la capacidad de destruir o crear geometría adicional. También puede pasar la información de los vértices a *buffers* de memoria usando la fase de *Stream Out*.
- **Rasterizer Stage:** procesa el color de los píxeles para las primitivas dadas.
- **Pixel Shader (PS):** coge "fragmentos" de pixeles y calcula el color de estos según lo que haya definido el programador. Hace labores de *clipping* (recortar fragmentos si estos están tapados por otra geometría). Es en esta fase donde se suelen hacer cálculos de iluminación.

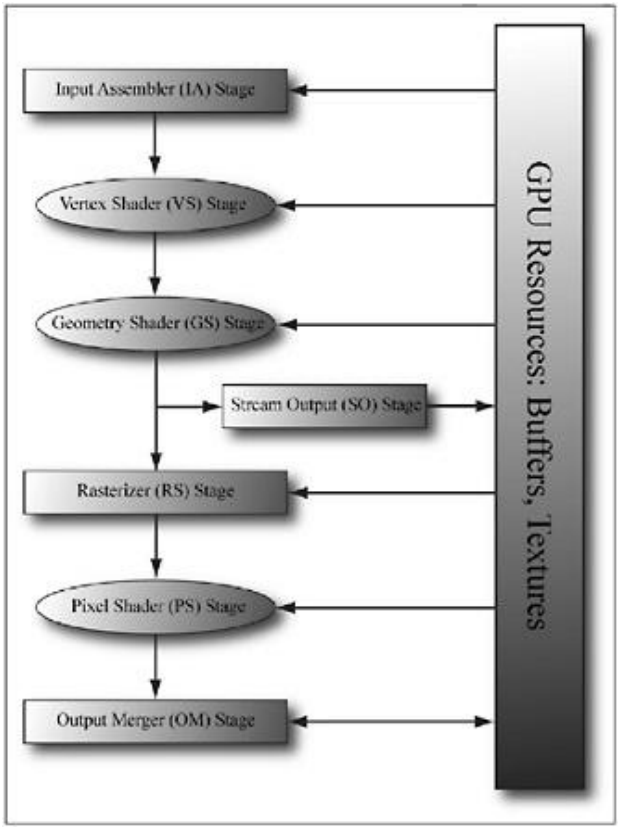

Fig. A.1 Diagrama de la *pipeline* gráfica

Todas estas fases están conectadas entre sí secuencialmente y algunas de ellas pueden tanto leer como escribir al buffer de memoria. Ver la figura A.1 para más claridad. Las fases que podemos definir internamente son la de *Vertex Shader*, *Geometry Shader* y *Pixel Shader*. Esto implica que podemos programar *shaders* con distinto comportamiento para aplicar a diferentes objetos.

A.2 Ruido

En el contexto de la teoría de señales, entendemos el ruido como una perturbación o interferencia. En el contexto de los gráficos por computador, hablamos de una primitiva de textura procedural generada por una función matemática. Estos algoritmos nos permiten crear patrones que recrean fielmente la naturaleza, alcanzando un mayor grado de realismo. Es por ello que nos serán de utilidad a la hora de modelar nuestro terreno y generar texturas con algo de interés y granularidad. A continuación se exponen los dos tipos de ruido que utilizaremos.

A.2.1 Ruido de Perlin y Simplex

El algoritmo de ruido más utilizado en gráficos por computador es el **ruido de Perlin**, creado por Ken Perlin en 1997. Dicho algoritmo es capaz de crear patrones de aspecto muy natural, por lo que es de gran utilidad para recrear efectos como el fuego, el humo o las nubes. Sin embargo, tiene un coste computacional alto a medida que lo calculamos en un número alto de dimensiones. Es por ello que Perlin ideó una nueva versión de su algoritmo en 2001, llamada **ruido Simplex** [8].

Aunque el algoritmo funciona para N dimensiones, nos centraremos en un caso concreto (2 dimensiones) para explicar el proceso. En el ruido clásico de Perlin, lo que hacemos es asignar una serie de gradientes (vectores de inclinación) pseudo-aleatorios a cada coordenada entera (0, 1, 2, 3…) y hacer que la función de interpolación valga 0 en dichas coordenadas. Para un punto *x* en el espacio, se obtiene el valor de ruido calculando el producto vectorial de los vectores de distancia entre *x* y los 2^N gradientes (donde N son las dimensiones) en los puntos de coordenadas enteras *Q* más cercanos, para a continuación interpolar dichos valores con una función base *f(t)*. En la figura A.2 podemos ver un ejemplo gráfico.

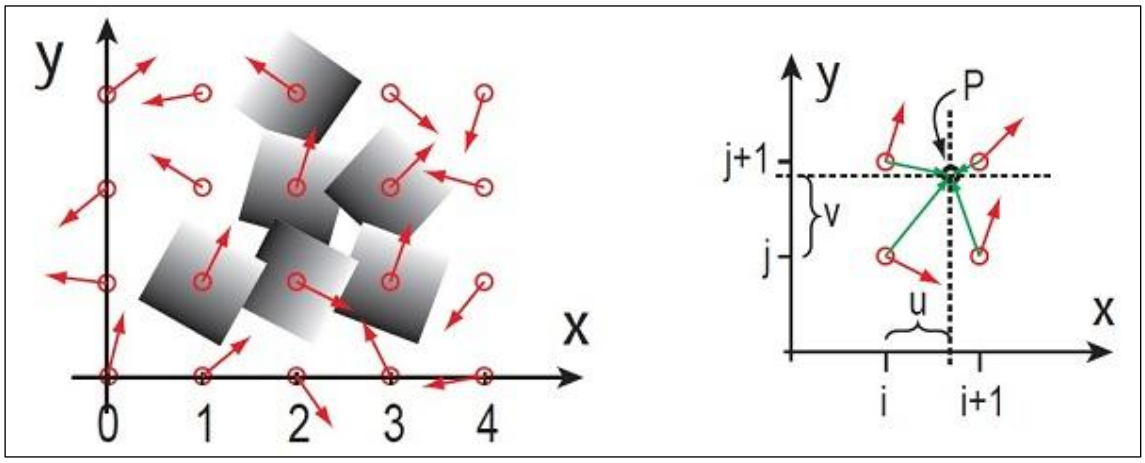

Fig. A.2 Cálculo de Ruido de Perlin en 2 dimensiones

Dados:

P = (x, y), i = min(x) , j = min(y)
g00 = gradiente en (i, j) g10 = gradiente en (i + 1, j)
g01 = gradiente en (i, j + 1) g11 = gradiente en (i + 1, j + 1)

$u = x - i, v = y - j$

$f(t) = 6t^5 - 15t^4 + 10t^3$

Se calcula:

$n_{00} = g_{00}\begin{bmatrix} u \\ v \end{bmatrix}, n_{10} = g_{10}\begin{bmatrix} u-1 \\ v \end{bmatrix}, n_{01} = g_{01}\begin{bmatrix} u \\ v-1 \end{bmatrix}, n_{11} = g_{11}\begin{bmatrix} u-1 \\ v-1 \end{bmatrix}$

$n_{x0} = n_{00}\big(1 - f(u)\big) + n_{10}f(u), n_{x1} = n_{01}\big(1 - f(u)\big) + n_{11}f(u)$

$n_{xy} = n_{x0}\big(1 - f(v)\big) + n_{x1}f(v)$

Donde n_{xy} es el valor final que nos devuelve la función de ruido.

Este sería el proceso para calcular el ruido de Perlin en 2 dimensiones. En el caso general, por cada N dimensión se calculan 2^N gradientes y 2^{N-1} interpolaciones.

Para la versión Simplex, lo que hacemos es coger la forma más compacta posible que sea repetible y llene el espacio disponible (por ejemplo, un triángulo equilátero para dos dimensiones). En general, para N dimensiones, dicha forma tendrá N+1 esquinas. A esto lo llamaremos una forma Simplex.

El problema del ruido de Perlin es que se calculan interpolaciones a lo largo de cada dimensión, lo cual aumenta el coste computacional. En el caso del Simplex lo que hacemos es sumar las contribuciones de cada esquina de nuestra forma, las cuales se calculan multiplicando la extrapolación de los gradientes con una función de atenuación radialmente simétrica (figura A.3).

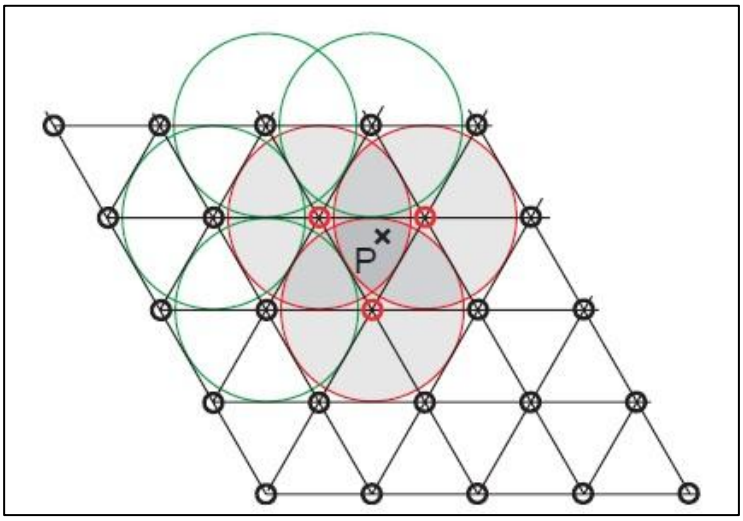

Fig. A.3 Simplex en 2D. Contribuciones en rojo

El único obstáculo a la hora de realizar esto es saber en qué Simplex concreto nos encontramos. Para ello, primero dividimos el eje de coordenadas en N! símplices que forman un híper cubo N-dimensional (figura A.4) y luego cogemos las coordenadas enteras de nuestro punto P para determinar el Simplex objetivo, comparando las magnitudes de las distancias desde el origen hasta P.

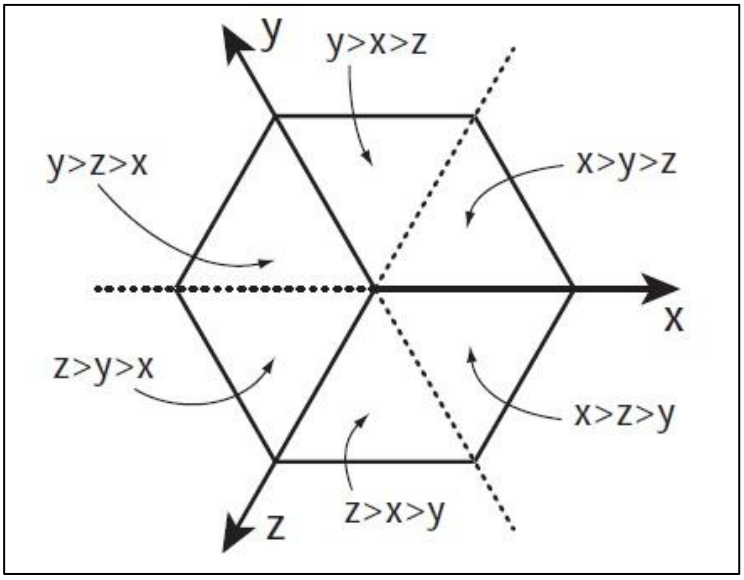

Fig. A.4 División del espacio en el Símplex 3D

En definitiva, el coste de cada versión del algoritmo es el siguiente:

- **Ruido de Perlin:** $O(2^N)$. Más eficiente para 1D, 2D y 3D.
- **Ruido Simplex:** $O(N^2)$. Más eficiente para 4D y superior.

A.2.2 Ruido de Voronoi

También conocido como **ruido de Worley** o simplemente **ruido celular**, se trata de otra primitiva de textura procedural creada por Steven Worley en 1996. El algoritmo crea una textura de consistencia celular, que viene determinada por una función de combinación y la ecuación de distancia que utilicemos para medir un punto *x* respecto a una serie de puntos característicos distribuidos aleatoriamente en el espacio [9]. En la figura A.5 veremos un ejemplo gráfico.

El algoritmo consiste en los siguientes pasos:

1. Dividimos el espacio en una rejilla de cuadros, con cada cuadro localizado en coordenadas enteras. Dado un punto A en dicho espacio, buscamos en qué cuadrante se encuentra.

2. A continuación debemos determinar cuántos puntos característicos se encuentran en nuestro cuadrante. Usando una distribución de Poisson, podemos calcular un número aleatorio entre 0 y 1 y compararlo con las probabilidades de la distribución para determinar el número de puntos.

3. Generamos las coordenadas para los puntos de característica y medimos la distancia de cada uno a A para saber cual está más próximo. También comprobamos los cuadrantes vecinos para asegurarnos que no hay puntos aún más cercanos.

4. Una vez encontrado el punto más cercano, el valor de la función de ruido es la distancia entre A y dicho punto, determinada por la métrica de distancia que utilicemos.

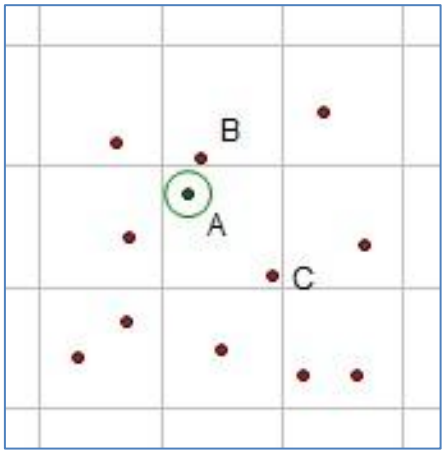

Fig. A.5 Ejemplo de búsqueda de puntos en ruido de Voronoi

En función de la métrica utilizada, la textura resultante tendrá dispares resultados. Podemos ver algunos de ellos en la figura A.6

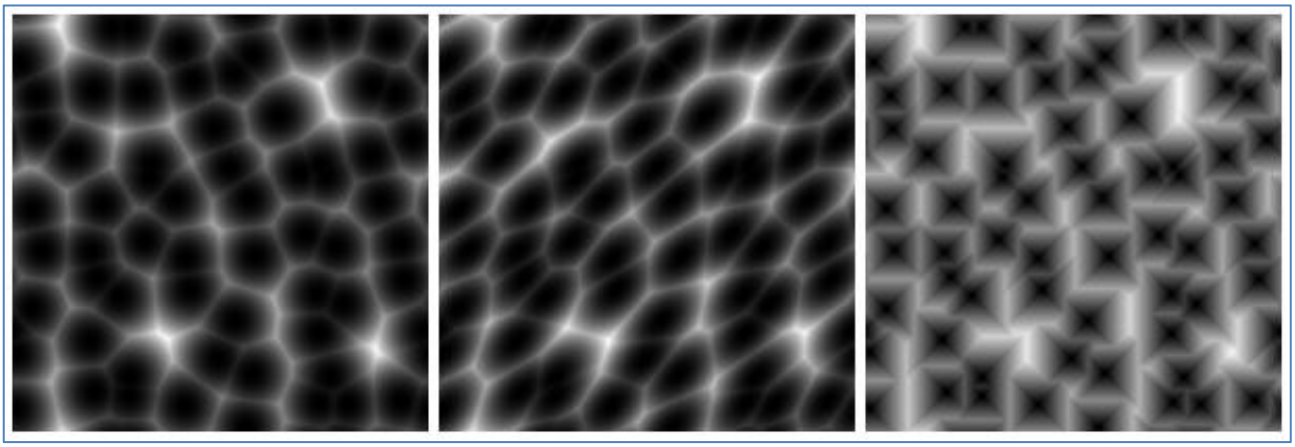

Fig. A.6 De izquierda a derecha: distancia euclídea, cuadrática y de Chebyshev

A.3 Representación gráfica de volúmenes

A la hora de representar un terreno 3D, podemos seguir diferentes acercamientos. Uno de los más conocidos son los **mapas de alturas**, o *height fields*. No son más que texturas (generadas a mano o proceduralmente) con la información de altura en cada punto, lo cual se aplica a una malla 3D plana para construir el terreno en cuestión. Algoritmos como el *Diamond-square*[9] (figura A.7) son usados generalmente para crear este tipo de mapas [10].

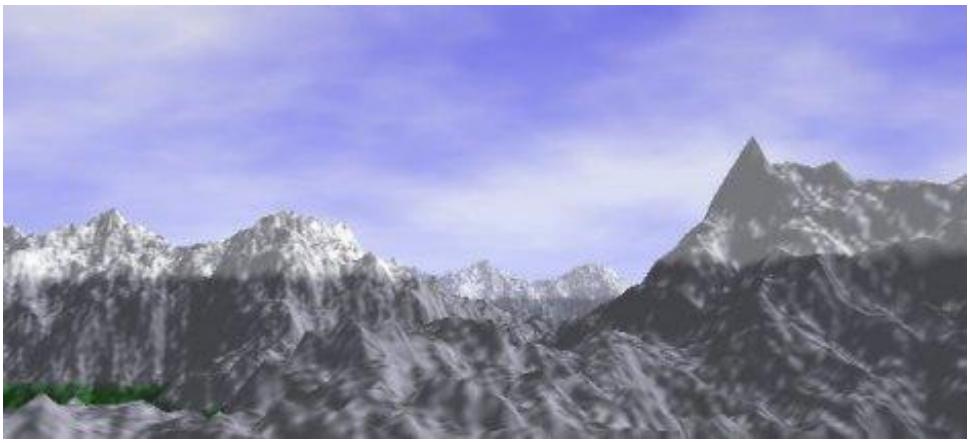

Fig. A.7 Ejemplo de terreno generado con el algoritmo *Diamond-Square*

Sin embargo, esta forma de generar terrenos tiene un problema de base: se está trabajando sobre dos dimensiones. Lo que se está haciendo, a grandes rasgos, es obtener una altura *z* a partir de una función f(*x* , *y*). Esto provoca que los resultados obtenidos acaben siendo siempre similares y limitados: montañas, picos y valles.

Para solucionar este problema, tendremos que trabajar con una función que acepte las tres coordenadas espaciales. Para ello, es necesario utilizar **vóxeles**, unidades de volumen que conforman una rejilla tridimensional. De esa forma, la función a utilizar se convierte en una **función de densidad**, la cual representa una superficie volumétrica.

Existe, no obstante, un problema con este acercamiento: los vóxeles no son fáciles de visualizar en pantalla. Existen maneras de hacerlo, pero simplemente las GPUs no están optimizadas para ello, lo cual repercute en el rendimiento. Una manera simple de solucionar el problema es representar cada vóxel como un cubo en el espacio, con valor de 0 o 1 según si se considera tierra o aire. Videojuegos como el famoso *Minecraft* (figura A.8) siguen este método.

[9] Ver http://en.wikipedia.org/wiki/Diamond-square_algorithm

Fig. A.8 Captura de *Minecraft*

Otra manera de acercarse al problema es no representar los vóxeles directamente, sino usarlos como una rejilla de muestreo para la función de densidad, la cual aproximaremos a una malla poligonal, mucho más amigable a la hora de ser procesada por una GPU. Esto nos permite generar terrenos con características más diversas e interesantes. A continuación, se explicarán los algoritmos usados para llevar esto a cabo.

A.3.1 Marching Cubes

Marching Cubes es un algoritmo creado en 1987 por William E. Lorensen y Harvey E. Cline [14]. Originalmente ideado para visualizar conjuntos de datos de dispositivos IRM[10], nos permite extraer una representación poligonal de la isosuperfície de un campo escalar 3D. Resulta un algoritmo sencillo y rápido, ya que se funciona enteramente con *look-up tables* [11]. En la época en que se creó no hubiera sido muy eficiente para representar terrenos relativamente grandes, pero con la potencia de las CPU y GPU de hoy en día resulta más que asequible.

El algoritmo consiste en lo siguiente: se divide el espacio en una rejilla rectangular 3D formada por cubos. Estos cubos tienen sus esquinas y aristas convenientemente indexadas, de forma que se puedan referenciar (figura A.9). A continuación, evaluaremos la función de densidad en cada una de las esquinas, comprobaremos los valores obtenidos y marcaremos los índices de las esquinas cuyo valor esté por encima de un límite que hayamos establecido previamente. Con eso obtendremos

[10] Imagen por Resonancia Magnética

un índice de 8 bits que utilizaremos para consultar la tabla de aristas o *edgeTable*, la cual consta de hasta 256 casos posibles.

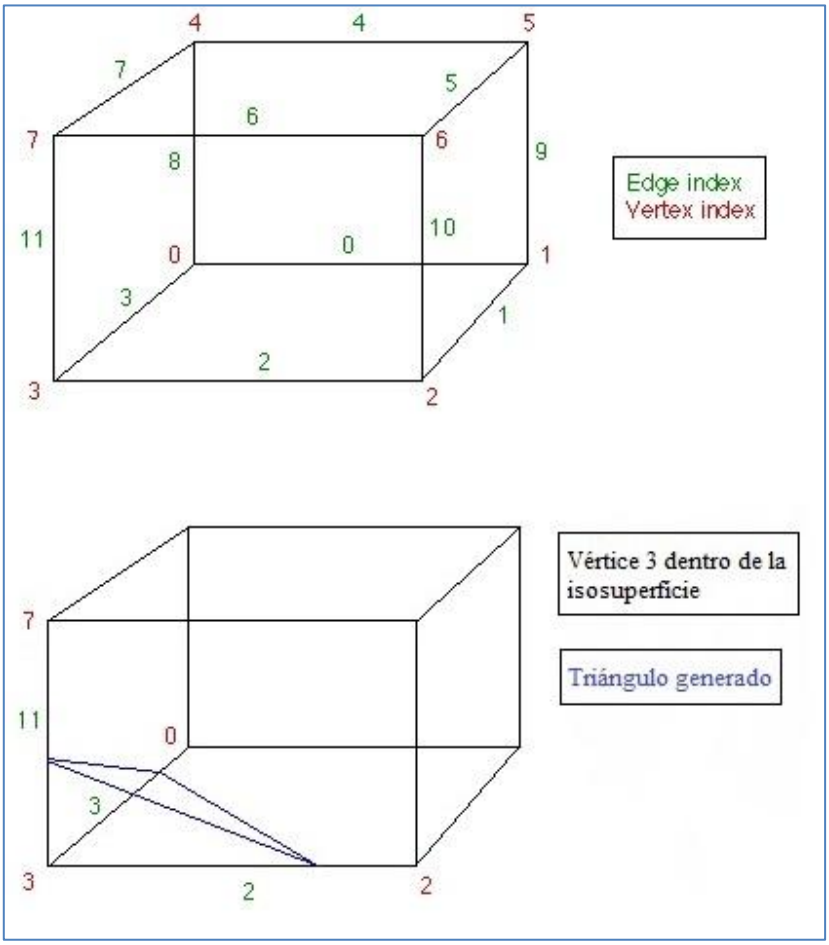

Fig. A.9 Arriba, un cubo con todos sus índices. Abajo, un caso específico

Al consultar dicha tabla, obtenemos una serie de 12 bits, cada uno correspondiendo a una arista del cubo (0 si no está cortada por la superficie, y 1 si lo está). Una vez se conocen las aristas atravesadas por la isosuperfície, se encuentran los puntos de intersección calculando una interpolación lineal entre los dos extremos de la arista con la siguiente fórmula. Si P_1 y P_2 equivalen a los puntos de la arista cortada y V_1 y V_2 a los valores de la función de densidad en esos puntos, el punto de intersección P equivale a:

$$P = P_1 + (límite - V_1) * (P_2 - P_1)/(V_2 - V_1)$$

Los puntos P calculados formarán un triángulo en la malla poligonal. Se consulta una última tabla (*triTable*) usando nuestro código de 12 bits para que nos indique en qué orden dibujar los vértices de los triángulos. Según el caso, podemos llegar a generar de 0 a 5 triángulos por cubo (figura A.10). Repetimos todo este proceso con cada cubo para generar nuestra malla aproximada a

la isosuperfície. Por supuesto, la calidad y fidelidad de esta aproximación será mayor cuanto mayor sea la resolución de rejilla de cubos que utilicemos.

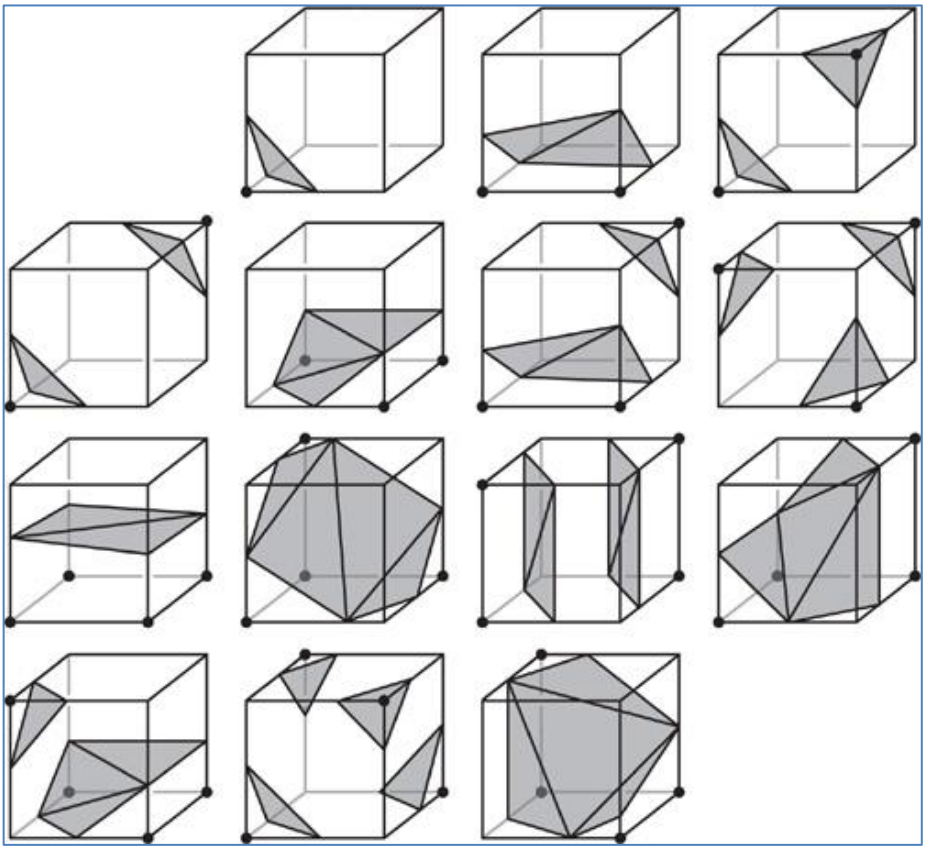

Fig. A.10 Diferentes casos posibles en el algoritmo *Marching Cubes*

A.3.2 Marching Tetrahedrons

Marching Tetrahedrons no es más que una revisión respecto al antiguo algoritmo de *Marching Cubes*. El funcionamiento es prácticamente idéntico, con la salvedad de que esta vez no utilizamos un cubo para hacer el muestreo de la superficie, sino que dividimos éste en 6 tetraedros irregulares, ejecutando el algoritmo en cada uno de ellos.

La principal ventaja de esta nueva versión es que reduce enormemente el número de casos a contemplar (8 con respecto a 256), dado que esta vez solo tenemos que comprobar 4 vértices por objeto (figura A.11). Además, elimina alguna de las ambigüedades que presentaba el algoritmo anterior en alguno de sus casos y mejorando la resolución de muestreo. Su única pega es que tendremos que procesar 6 tetraedros donde antes solo procesábamos un cubo, aumentando así el número de objetos a iterar y los requerimientos de memoria.

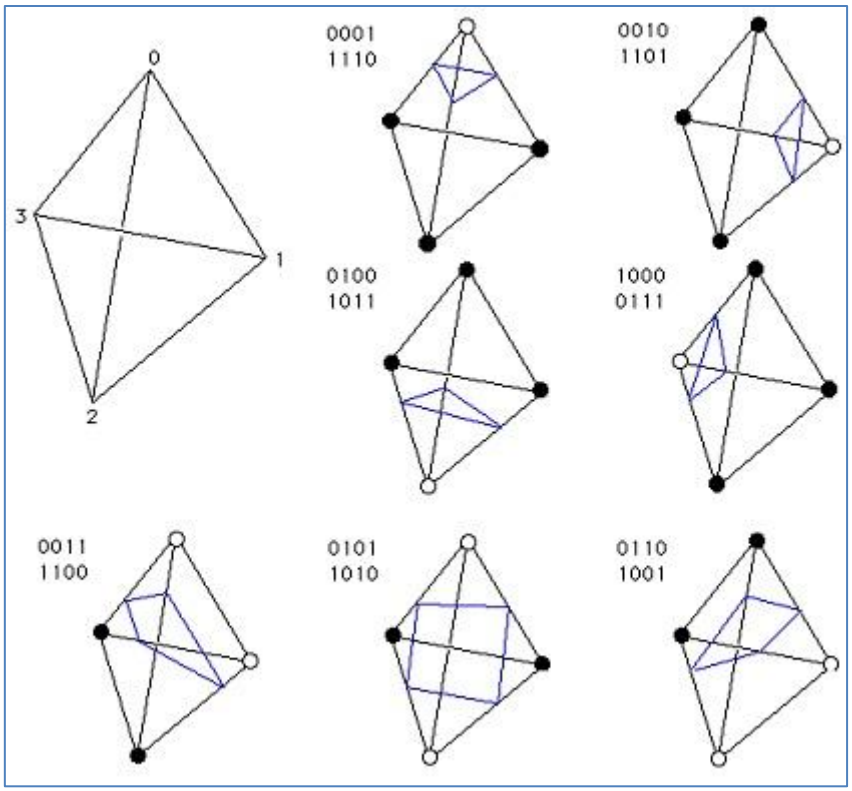

Fig. A.11 Los 8 posibles casos de *Marching Tetrahedons*

Anexo B

Manual de usuario

En este anexo se definen las posibles acciones a ejecutar en la demo, las cuales funcionan enteramente con el teclado. Aunque la interactividad sea limitada, hay unos cuantos comandos a tener en cuenta:

- **W-A-S-D:** Movimiento y rotación de la cámara en el plano XZ.
- **R-F:** Sube y baja la cámara en el eje Y.
- **Q-E:** Baja o sube la vista de la cámara.
- **T:** Activa/desactiva el modo "alambre" (vemos los polígonos sin color). Está desactivado por defecto.
- **L:** Activa/desactiva el *overlay* de texto (figura B.1) que muestra información de los FPS, el tiempo que transcurre entre cada *frame* mostrado y la posición de la cámara en coordenadas mundo, así como una lista de los posibles comandos a usar.
- **I:** Cambia la semilla del generador de números aleatorios, con lo que los valores de la función de ruido para el terreno cambian, provocando que cambie su forma en tiempo real.
- **P:** Activa/desactiva el bloqueo de la cámara al bloque de terreno, de forma que al estar desactivado podemos movernos sin que el terreno se genere alrededor nuestro.

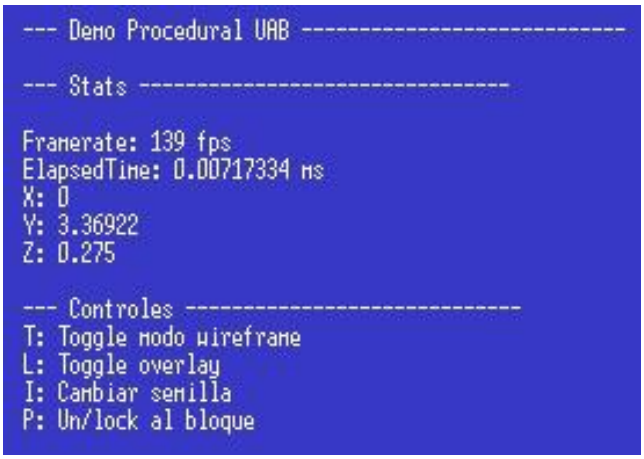

Fig. B.1 Captura del *overlay* de texto

Firmado: David Rodríguez Real

Bellaterra, 19 de junio de 2012

Resum

Aquest projecte consisteix en el desenvolupament d'una demo 3D utilitzant exclusivament gràfics procedurals per tal d'avaluar la seva viabilitat en aplicacions més complexes com els videojocs. En aquesta aplicació es genera un terreny aleatori explorable amb vegetació i textures creades proceduralment.

Resumen

Este proyecto consiste en el desarrollo de una demo 3D utilizando exclusivamente gráficos procedurales con el objetivo de evaluar su viabilidad en aplicaciones más complejas como videojuegos. En esta aplicación se genera un terreno aleatorio explorable con vegetación y texturas creadas proceduralmente.

Abstract

This project consists in developing a 3D demo using exclusively procedural graphics to evaluate their viability on more complex applications, like videogames. In this particular application, a random, explorable terrain is generated, along with procedural vegetation and textures.